1. 基本概念

布隆过滤器（Bloom Filter）是1970年由布隆提出的。它实际上是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索一个元素是否在一个集合中。它的优点是空间效率和查询时间都远远超过一般的算法，缺点是有一定的误识别率和删除困难。

Google爬虫 它要判断。哪些网页是被爬过来了的。

如果想要判断一个元素是不是在一个集合里，一般想到的是将所有元素保存起来，然后通过比较确定。链表，树等等数据结构都是这种思路. 但是随着集合中元素的增加，我们需要的存储空间越来越大，检索速度也越来越慢(O(n),O(logn))。不过世界上还有一种叫作散列表（又叫哈希表，Hash table）的数据结构（有一个动态数组，+ 一个hash函数）。它可以通过一个Hash函数将一个元素映射成一个位阵列（Bit array）中的一个点。这样一来，我们只要看看这个点是不是1就可以知道集合中有没有它了。这就是布隆过滤器的基本思想。

Hash面临的问题就是冲突。假设Hash函数是良好的，如果我们的位阵列长度为m个点，那么如果我们想将冲突率降低到例如 1%, 这个散列表就只能容纳m / 100个元素。显然这就不叫空间效率了（Space-efficient）了。解决方法也简单，就是使用多个Hash，如果它们有一个说元素不在集合中，那肯定就不在。如果它们都说在，虽然也有一定可能性它们在说谎，不过直觉上判断这种事情的概率是比较低的。

Bloom Filter是一种空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。Bloom Filter的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter通过极少的错误换取了存储空间的极大节省

总结起来说：bloomfilter，布隆过滤器：迅速判断一个元素是不是在一个庞大的集合内，但是他有一个弱点：它有一定的误判率

误判率：原本不存在于该集合的元素，布隆过滤器有可能会判断说它存在，但是，如果布隆过滤器判断说某一个元素不存在该集合，那么该元素就一定不在该集合内

2. 优缺点

2.1. 优点

相比于其它的数据结构，布隆过滤器在空间和时间方面都有巨大的优势。布隆过滤器存储空间和插入/查询时间都是常数。另外, Hash函数相互之间没有关系，方便由硬件并行实现。布隆过滤器不需要存储元素本身，在某些对保密要求非常严格的场合有优势。

布隆过滤器可以表示全集，其它任何数据结构都不能；

k和m相同，使用同一组Hash函数的两个布隆过滤器的交并差运算可以使用位操作进行。

```
1、能快速的判断元素存在不存在
2、远远的缩小存储数据的规模
```

2.2. 缺点

但是布隆过滤器的缺点和优点一样明显。误算率是其中之一。随着存入的元素数量增加，误算率随之增加。但是如果元素数量太少，则使用散列表足矣。

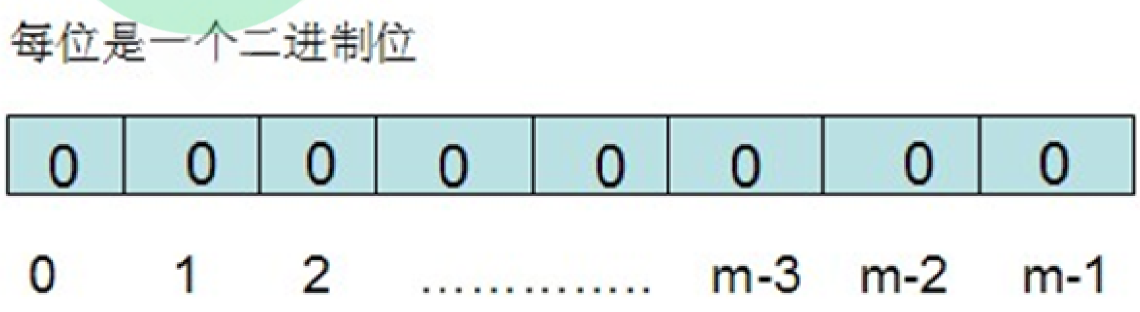
另外，一般情况下不能从布隆过滤器中删除元素。我们很容易想到把位列阵变成整数数组，每插入一个元素相应的计数器加1, 这样删除元素时将计数器减掉就可以了。然而要保证安全的删除元素并非如此简单。首先我们必须保证删除的元素的确在布隆过滤器里面. 这一点单凭这个过滤器是无法保证的。另外计数器回绕也会造成问题。在降低误算率方面，有不少工作，使得出现了很多布隆过滤器的变种。

```
1、存在一定的误判率，那么在你不能容忍有错误率的情况，布隆过滤器不适用
2、布隆过滤器不支持删除操作
```

3. 实现原理

布隆过滤器需要的是一个位数组(和位图类似, bytes数组)和K个映射函数(和Hash表类似)，在初始状态时，对于长度为m的位数组array，它的所有位被置0。

“huangbo”.hashCode() => 3



对于有n个元素的集合S={S1,S2...Sn},通过k个映射函数{f1,f2,......fk}，将集合S中的每个元素Sj(1<=j<=n)映射为K个值{g1,g2...gk}，然后再将位数组array中相对应的array[g1],array[g2]......array[gk]置为1：

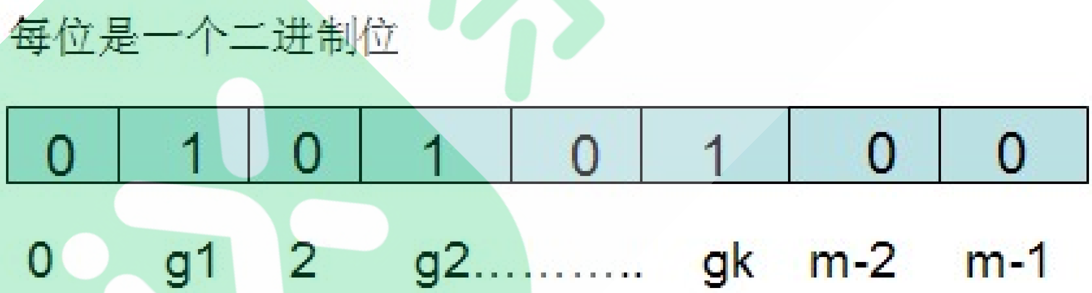
前两次是插入，插入的数据是“huangbo”“wangbaoqiang”

```
“huangbo”.hashCode1() => 2
“huangbo”.hashCode2() => 4
“wangbaoqiang” .hashCode1() => 3
“wangbaoqiang” .hashCode2() => 4

查询：“xuzheng”
“xuzheng”. hashCode1() => 4 有可能存在，有可能不存在，就算全部结果都是1，也有可能不存在
“xuzheng”. hashCode2() => 1 能百分百确定，这个值一定不存在
只要所有的hash函数计算出来的值里面有一个值是0 ，，我们就能确定这个key = “xuzheng”他一定不存在与我们的集合里面
```

自定义一个布隆过滤器的时候需要做的事情：

```
1、  初始化一个位数组
2、  实现K个hash函数
3、  实现查询，和 插入操作

查询和插入操作需要做的事情：
对插入进来的值进行hash计算，有几个hash函数，就计算几次，每次计算出来的结果值，都根据这个值，去位数组里面把相应位置的0变成1
对查询操作来说，只需要把你要查询的这个key值进行k个hash函数的调用，然后再判断计算出来的这个k个值对应的维数组上的值是不是有一个为0,，，如果有一个为0，那就表示，该key不在这个集合里面
```

N个hashcode()，N位的0变成1



要查找某个元素item是否在S中，则通过映射函数{f1,f2,...fk}得到k个值{g1,g2...gk}，然后再判断array[g1],array[g2]...array[gk]是否都为1，若全为1，则item在S中，否则item不在S中。这个就是布隆过滤器的实现原理。

前面说到过，布隆过滤器会造成一定的误判，因为集合中的若干个元素通过映射之后得到的数值恰巧包括g1,g2,...gk，在这种情况下可能会造成误判，但是概率很小。

4. 哈希函数/哈希表

4.1. 概念

哈希表中元素是由哈希函数确定的。将数据元素的关键字K作为自变量，通过一定的函数关系（称为哈希函数），计算出的值，即为该元素的存储地址，也即一个元素在哈希表中的位置是由哈希函数决定的

4.2. 特点

1、如果两个散列值是不相同的（根据同一函数），那么这两个散列值的原始输入也是不相同的。

2、散列函数的输入和输出不是唯一对应关系的，如果两个散列值相同，两个输入值很可能是相同的。但也可能不同，这种情况称为 “散列碰撞”（或者 “散列冲突”）。

4.3. 哈希构造方法

1、 直接定址法

2、 数字分析法

3、 平方取中法

4、 折叠法

5、 除留余数法

6、 随机数法

4.4. 哈希碰撞

概念：即两个不同的关键字，通过同一个哈希函数计算得出的结果值一样的。

4.5. 解决哈希碰撞

1、 拉链法

拉出一个动态链表代替静态顺序存储结构，可以避免哈希函数的冲突，不过缺点就是链表的设计过于麻烦，增加了编程复杂度。此法可以完全避免哈希函数的冲突。

2、 多哈希法

设计二种甚至多种哈希函数，可以避免冲突，但是冲突几率还是有的，函数设计的越好或越多都可以将几率降到最低（除非人品太差，否则几乎不可能冲突）。

3、 开放地址法

开放地址法有一个公式：Hi=(H(key)+di) MOD m i=1,2,...,k(k<=m-1)

其中，m为哈希表的表长。di 是产生冲突的时候的增量序列。如果di值可能为1,2,3,...m-1，称线性探测再散列。

如果di取1，则每次冲突之后，向后移动1个位置.如果di取值可能为1,-1,4,-4,9,-9,16,-16,...k*k*,-*k*k(k<=m/2)

称二次探测再散列。如果di取值可能为伪随机数列。称伪随机探测再散列。

4、 建域法

假设哈希函数的值域为[0,m-1]，则设向量HashTable[0..m-1]为基本表，另外设立存储空间向量OverTable[0..v]用以存储发生冲突的记录。

5. 误判率估计

现在我们了解了布隆过滤器的大致工作原理了，那我们就来计算一下这个误判率

```
数组的大小：m
总共的数据大小为：n
hash函数的个数为：k
```

假设布隆过滤器中的hash function（哈希函数）满足simple uniform hashing（单一均匀散列）假设：每个元素都等概率地hash到m个slot中的任何一个，与其它元素被hash到哪个slot无关。若m为bit数，则:

对某一特定bit位在一个元素，调用了某个hash函数之后，别改成了1的概率是:

$$\frac{1}{m}$$

对某一特定bit位在一个元素由某特定hash function插入时没有被置位为1的概率为:

$$1-\frac{1}{m}$$

则k个hash function中没有一个对其置位为1的概率为，也就是该bit位在 k 次hash之后还一直保持为0的概率:

$$(1-\frac{1}{m})^k$$

如果插入了n个元素，但都未将其置位的概率为，也就是当所有的元素都被插入进来以后，某一个特定的bit位还没有被改成1的概率:

$$(1-\frac{1}{m})^{kn}$$

则此位置被置为1（被改成了1）的概率为，也就是当所有的元素都被插入进来以后，某一个特定的bit位被改成1的概率:

$$1-(1-\frac{1}{m})^{kn}$$

现在考虑查询阶段，若对应某个待查询元素的k bits全部置位为1。因此将某元素误判的概率为:

当要查询一个元素的时候，我们要调用k个hash函数得到k个位置值，然后再去判断这k个位置的位值是不是都是1:

$$\text{计算出来的}k\text{个位置值都是}1\text{的概率}=(1-(1-\frac{1}{m})^{kn})^k$$

假如说，没有误判率，那我们判断数据存在不存在的标准是不是就检查一下，所有的位置值都是1 就OK了。

因为当 $x\to 0$的时候，$(1+x)^{\frac{1}{x}}\approx e$，稍微做一下化简：2.71828...

最后:

$$(1-(1-\frac{1}{m})^{kn})^k = (1-(1-\frac{1}{m})^{-m\cdot\frac{-kn}{m}})^k \approx (1-e^{\frac{-kn}{m}})^k$$

从上式中可以看出，当m增大或n减小时，都会使得误判率减小，这也符合直觉。

6. 最优哈希个数

由上面计算出的结果，现在计算对于给定的m和n，k为何值时可以使得误判率最低。设误判率为k的函数为：

$$f(k) = (1-e^{-\frac{kn}{m}})^k$$

翻译一下，也就是当m和n确定了以后，我们应该设置k为多少能使误判率最低呢？

当确定了m和n之后，我们要求出一个k使f(k)的值最小

我们可以确定k,m,n三者之间的关系之后，我们可以保证误判率最小

首先，设

$$b = e^{\frac{n}{m}}$$

则上面的式子化简为：

$$f(k) = (1-b^{-k})^k$$

对两边都取对数，得出：

$$\ln f(k) = k\cdot\ln(1-b^{-k})$$

两边对k求导，得出：

$$\frac{1}{f(k)}\cdot f'(k) = \ln(1-b^{-k}) + k\cdot\frac{1}{1-b^{-k}}\cdot(-b^{-k})\cdot\ln b\cdot -1$$

$$\frac{1}{f(k)}\cdot f'(k) = \ln(1-b^{-k}) + k\cdot\frac{b^{-k}\cdot\ln k}{1-b^{-k}}$$

接着，来求最值：

$$\begin{gathered}
ln(1-b^{-k}) + k\cdot\frac{b^{-k}\cdot\ln k}{1-b^{-k}} = 0 \\
(1-b^{-k})\cdot\ln(1-b^{-k}) = -k\cdot b^{-k}\cdot\ln b \\
(1-b^{-k})\cdot\ln(1-b^{-k}) = b^{-k}\cdot\ln(b^{-k}) \\
1-b^{-k} = b^{-k} \\
b^{-k} = \frac{1}{2}
\end{gathered}$$

所以：

$$e^{-\frac{kn}{m}} = \frac{1}{2}$$

所以：

$$\frac{kn}{m} = \ln 2$$

所以：

$$k = \ln 2 \cdot \frac{m}{n} \approx 0.7 \cdot \frac{m}{n}$$

因此，即当 $k \approx 0.7 \cdot \frac{m}{n}$ 时， 误判率是最低的。

此时的误判率：

$$P(error) = (1 - \frac{1}{2})^k = 2^{-k} = 2^{-\ln 2 \cdot \frac{m}{n}} \approx 0.6185^{\frac{m}{n}}$$

可以看出若要使得误判率≤1/2，则：

$$k \geqslant 1 \Longrightarrow \frac{m}{n} \geqslant \frac{1}{\ln 2}$$

这说明了若想保持某固定误判率不变，布隆过滤器的bit数m与被add的元素数n应该是线性同步增加的

假设：

```
n = 100w数据
m = 2000w位数组大小
```

那么

$$k = 0.6185^{\frac{m}{n}} = 0.6185^{20} \approx= 0.000067$$

7. 总结

在计算机科学中，我们常常会碰到时间换空间或者空间换时间的情况，即为了达到某一个方面的最优而牺牲另一个方面。Bloom Filter在时间空间这两个因素之外又引入了另一个因素：错误率。在使用Bloom Filter判断一个元素是否属于某个集合时，会有一定的错误率。也就是说，有可能把不属于这个集合的元素误认为属于这个集合（False Positive），但不会把属于这个集合的元素误认为不属于这个集合（False Negative）。在增加了错误率这个因素之后，Bloom Filter通过允许少量的错误来节省大量的存储空间。

自从Burton Bloom在70年代提出Bloom Filter之后，Bloom Filter就被广泛用于拼写检查和数据库系统中。近一二十年，伴随着网络的普及和发展，Bloom Filter在网络领域获得了新生，各种Bloom Filter变种和新的应用不断出现。可以预见，随着网络应用的不断深入，新的变种和应用将会继续出现，Bloom Filter必将获得更大的发展。